Organization of Digital Computers Lab
EECS 112L

Lab 4: Pipelined Processor

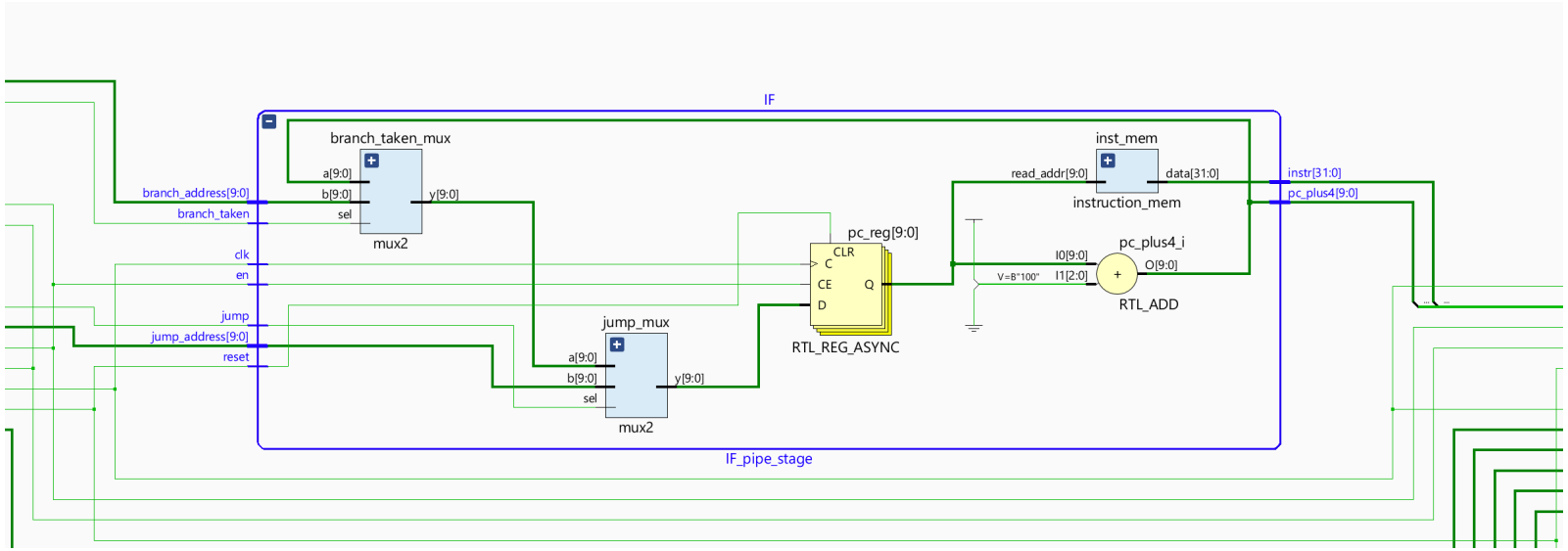
Armand Ahadi-Sarkani
ID # 60913515
March 8, 2020

Objective

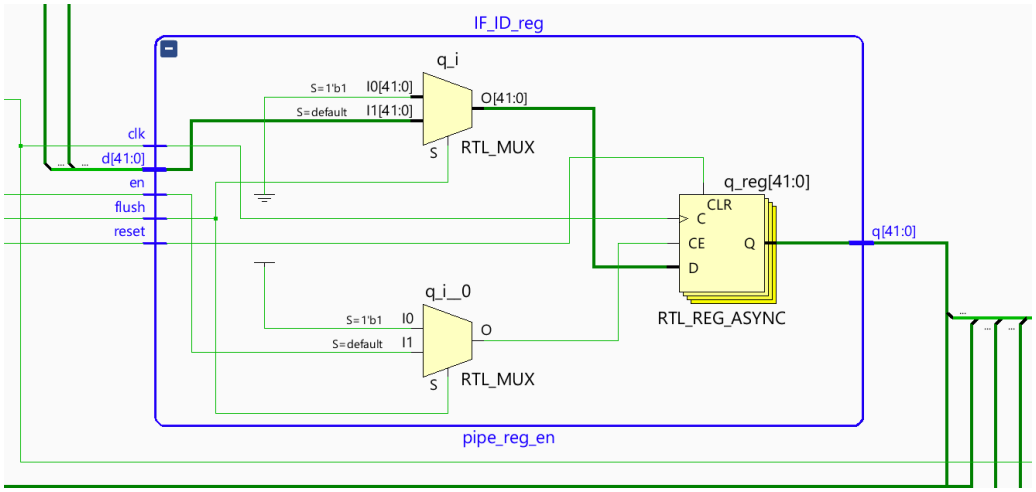
The objective of this lab is to turn the 32-bit MIPS processor created in Lab 2 into a pipelined processor, by breaking up the data path into five pipeline stages and intermediate registers. These five stages are the **instruction fetch** stage, where an instruction is fetched from memory, the **instruction decode** stage, where the registers in the instruction are read and decoded, the **execution** stage, where the operation is actually executed by the ALU, and the **write back stage,** to write the result of the instruction back into a register. There are intermediate registers that serve as buffers to transfer data between these stages of the pipeline. Pipelining will increase throughput, but inherently cause certain hazards, including structural, data, and control hazards, which will be dealt with by either stalling the pipeline, or forwarding data from a previous instruction.

Procedure

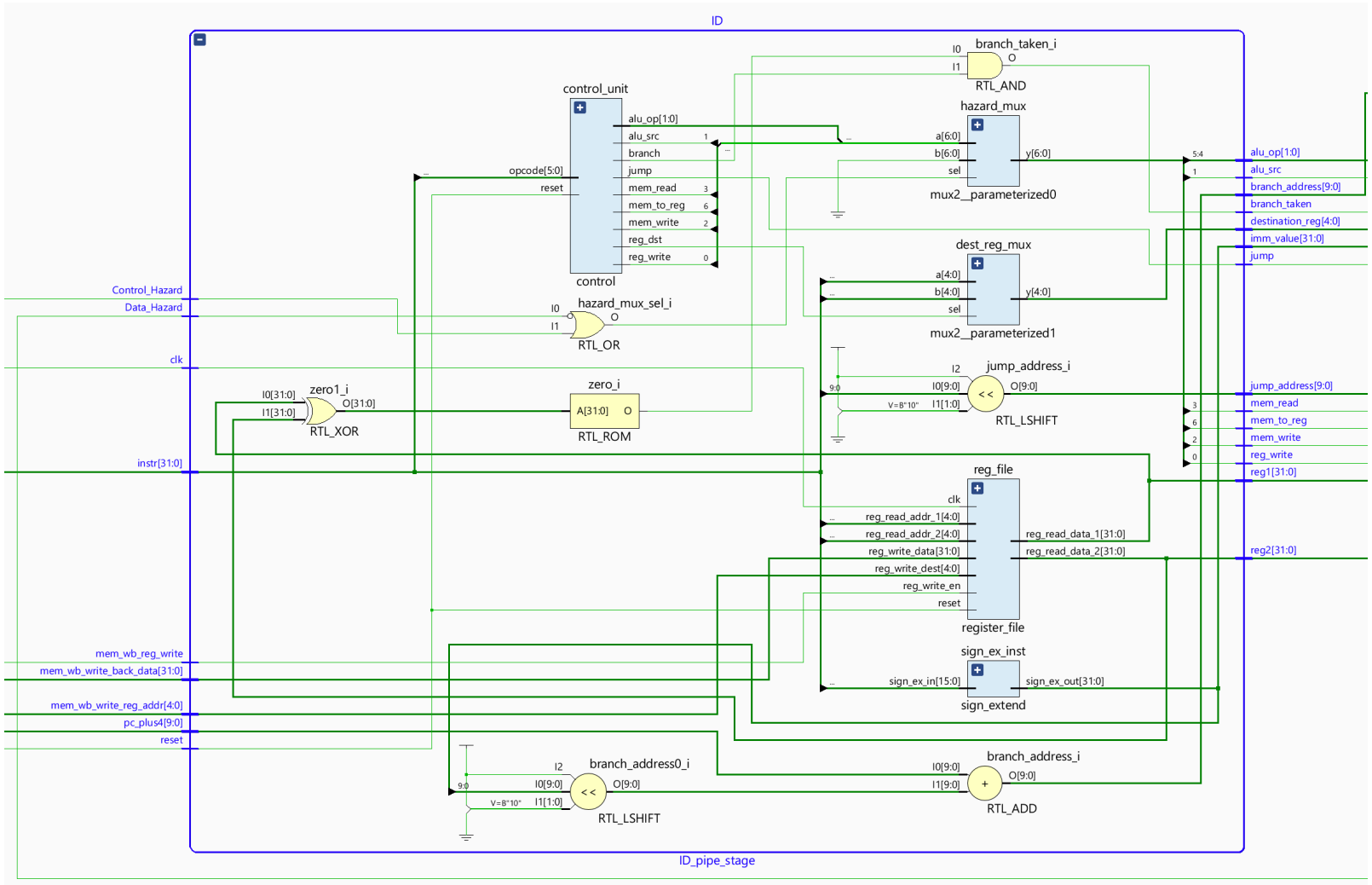
1. This lab was first approached by using the existing single cycle processor from Labs 2 and 3, and making some modifications to make room for the new pipelined datapath. The existing single datapath module was replaced by the five pipeline stages and their corresponding registers in the **mips32.v** file.
2. The first stage, **instruction fetch (IF)**, was implemented by modeling the schematic in the lab manual and re-wiring the instruction memory module here and instantiating several new multiplexers to handle the input and output data. The resultant detailed RTL schematic of this stage is shown below:



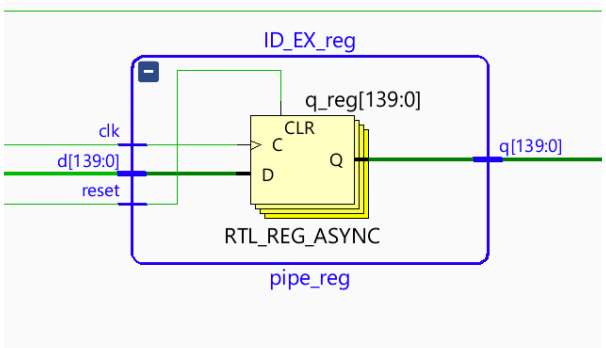
3. Next, the **IF/ID register** was implemented. This register helps transmit data as a buffer between the IF stage and the ID stage, and unlike other registers in this pipelined processor, it is driven by an asynchronous reset enable signal from the Data Hazard output of the hazard detection module. Multiple inputs were concatenated into one input (d), and the one output (q) was then split back into wires of the same bit-length as its inputs for use in future stages. The resultant schematic of this register is shown below:



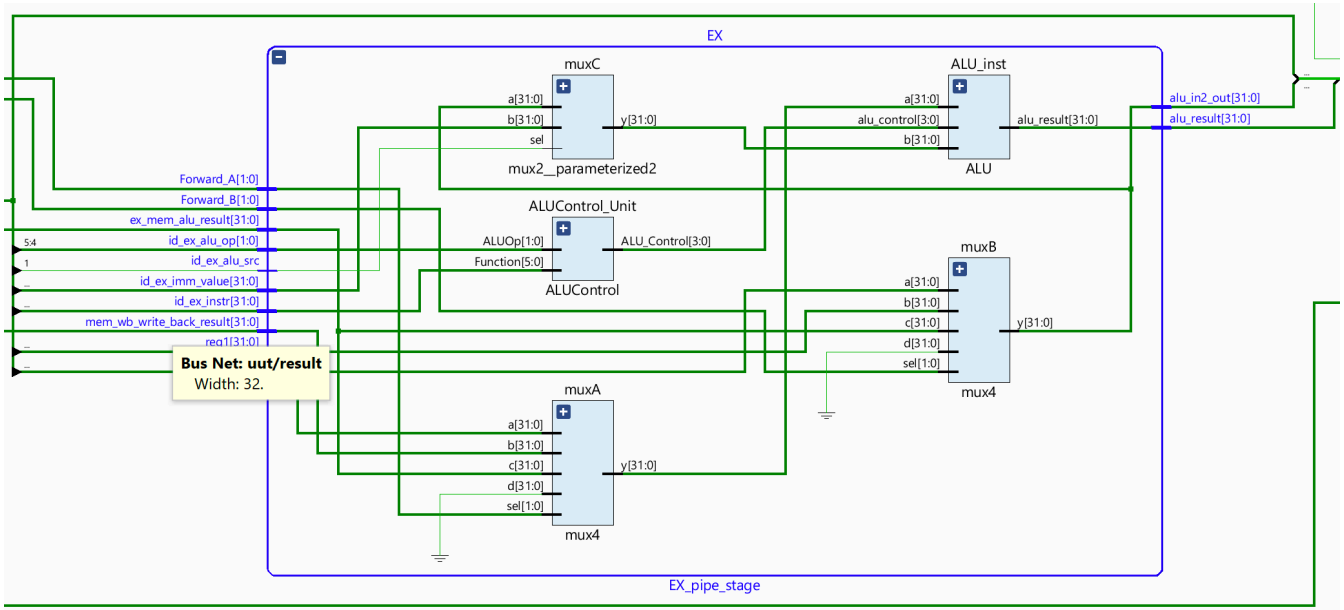
4. Next, the **instruction decode (ID)** pipeline stage was implemented by modeling the schematic in the lab manual and rewiring the control unit, register file, sign extend module, and multiple multiplexers here. The resultant schematic is shown below:



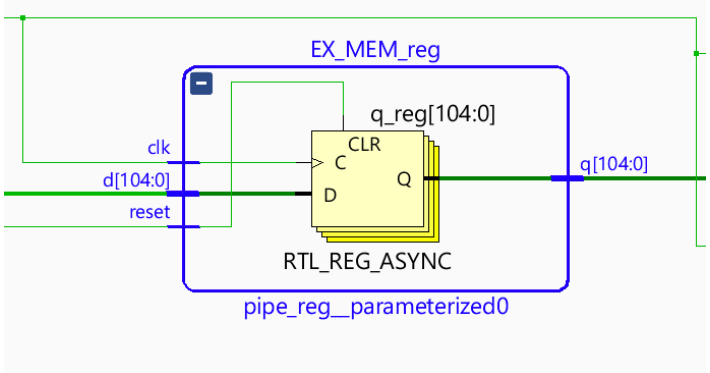
5. Next, the **ID/EX register** was implemented. This register transmits data through an ordinary buffer from the ID stage to the EX stage. Multiple inputs were concatenated into one input (d), and the one output (q) was then split back into wires of the same bit-length as its inputs for use in future stages. The resultant schematic is shown below:



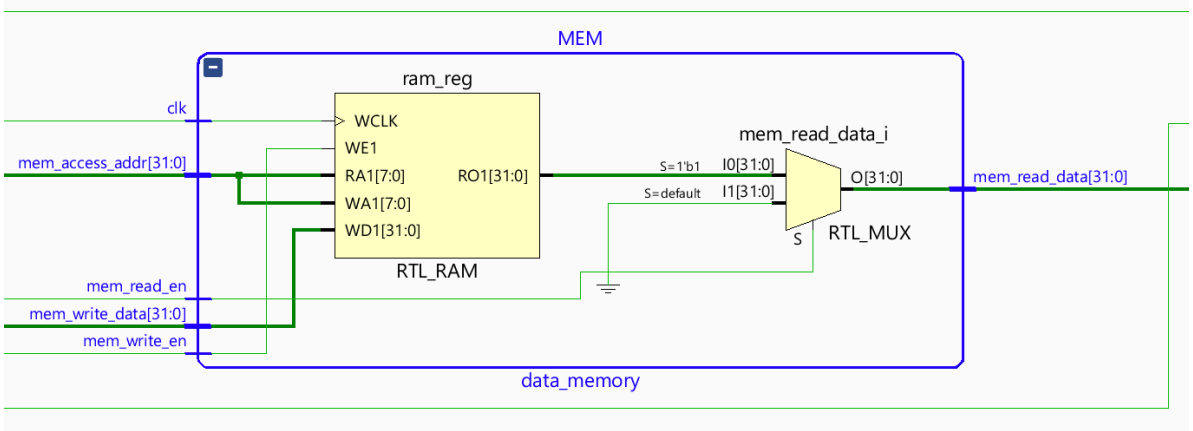
6. Next, the **execution (EX)** stage was implemented. This stage instantiates the ALU, ALU Control, and (new) 4-input and 2-input multiplexers. The resultant schematic is shown below:



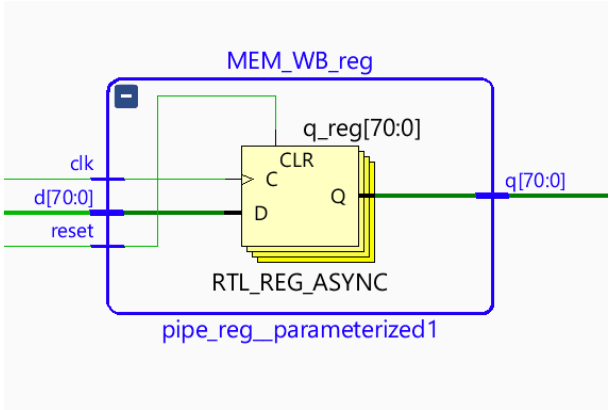
7. Next, the **EX/MEM register** was implemented. This register transmits data through an ordinary buffer from the EX stage to the MEM stage. Similarly to the other registers, multiple inputs were concatenated into one input (d), and the one output (q) was then split back into wires of the same bit-length as its inputs for use in future stages. The resultant schematic is shown below:



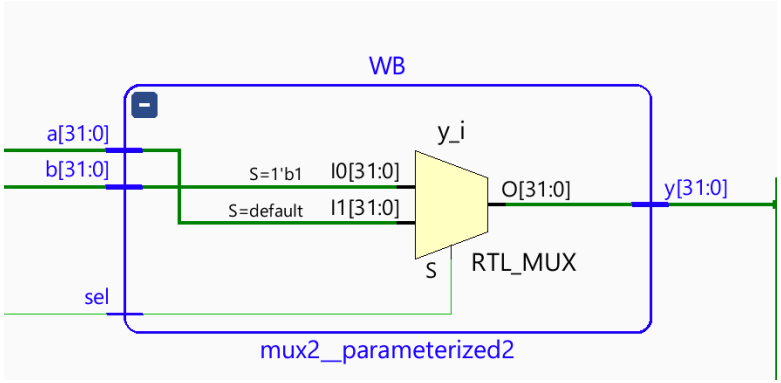
8. Next, the **memory (MEM)** stage was implemented. This stage is simply the same data memory module from Lab 2. The resultant schematic is shown below:



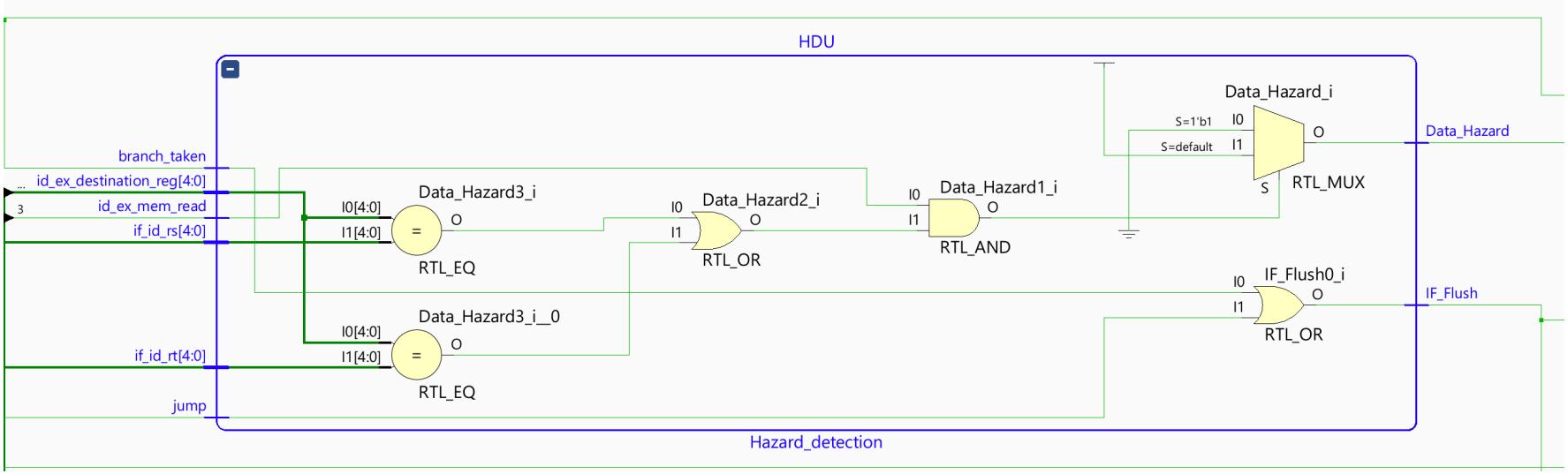
9. Next, the **MEM/WB register** was implemented. This register transmits data through an ordinary buffer from the MEM stage to the WB stage. Similarly to the other registers, multiple inputs were concatenated into one input (d), and the one output (q) was then split back into wires of the same bit-length as its inputs for use in future stages. The resultant schematic is shown below:



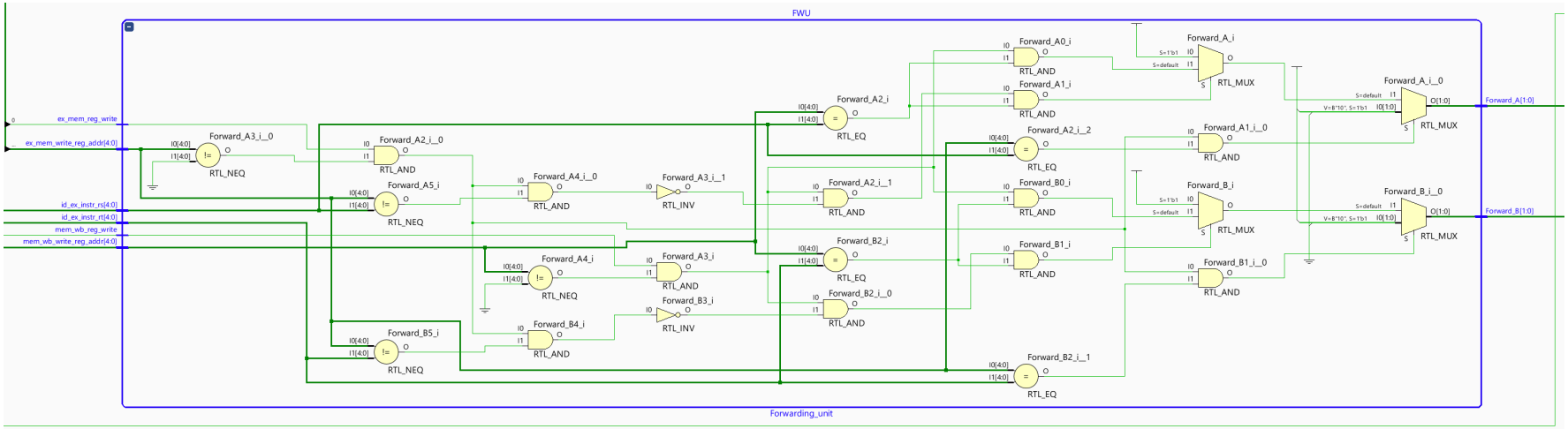
10. Next, the **write back (WB)** stage was implemented. This stage is simply a multiplexer selecting between two different options for write back. The resultant schematic is shown below:



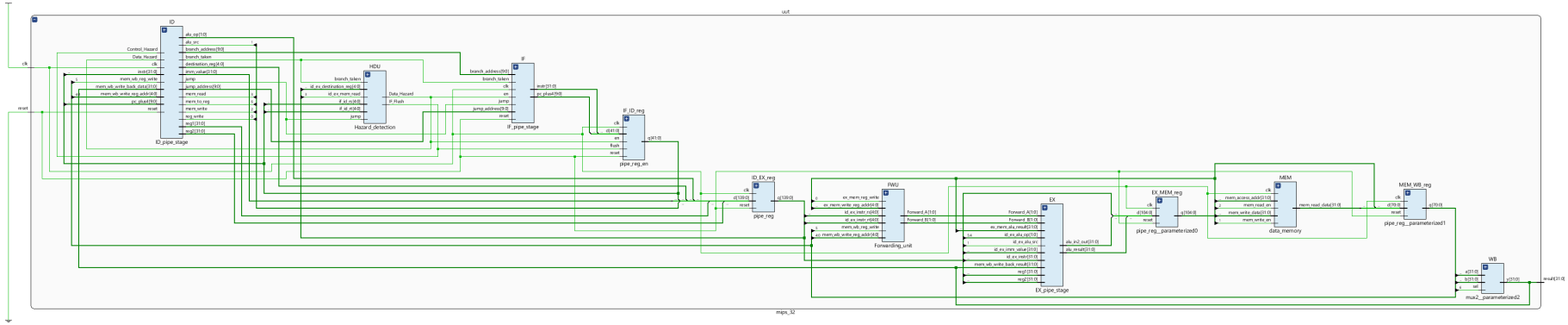
11. The **hazard detection** module was also implemented, in order to insert stalls into the pipeline in case of a data hazard or control hazard (in cases where forwarding will not work). The resultant schematic is shown below:



12. The **forwarding unit** was created to allow for forwarding instead of stalls in the case of a data dependency on a previous instruction. The schematic is shown below:



13. A high-level schematic is shown below for the entire processor, showing all of the connections between the different stages and modules (zoom for detail):



Simulation Results

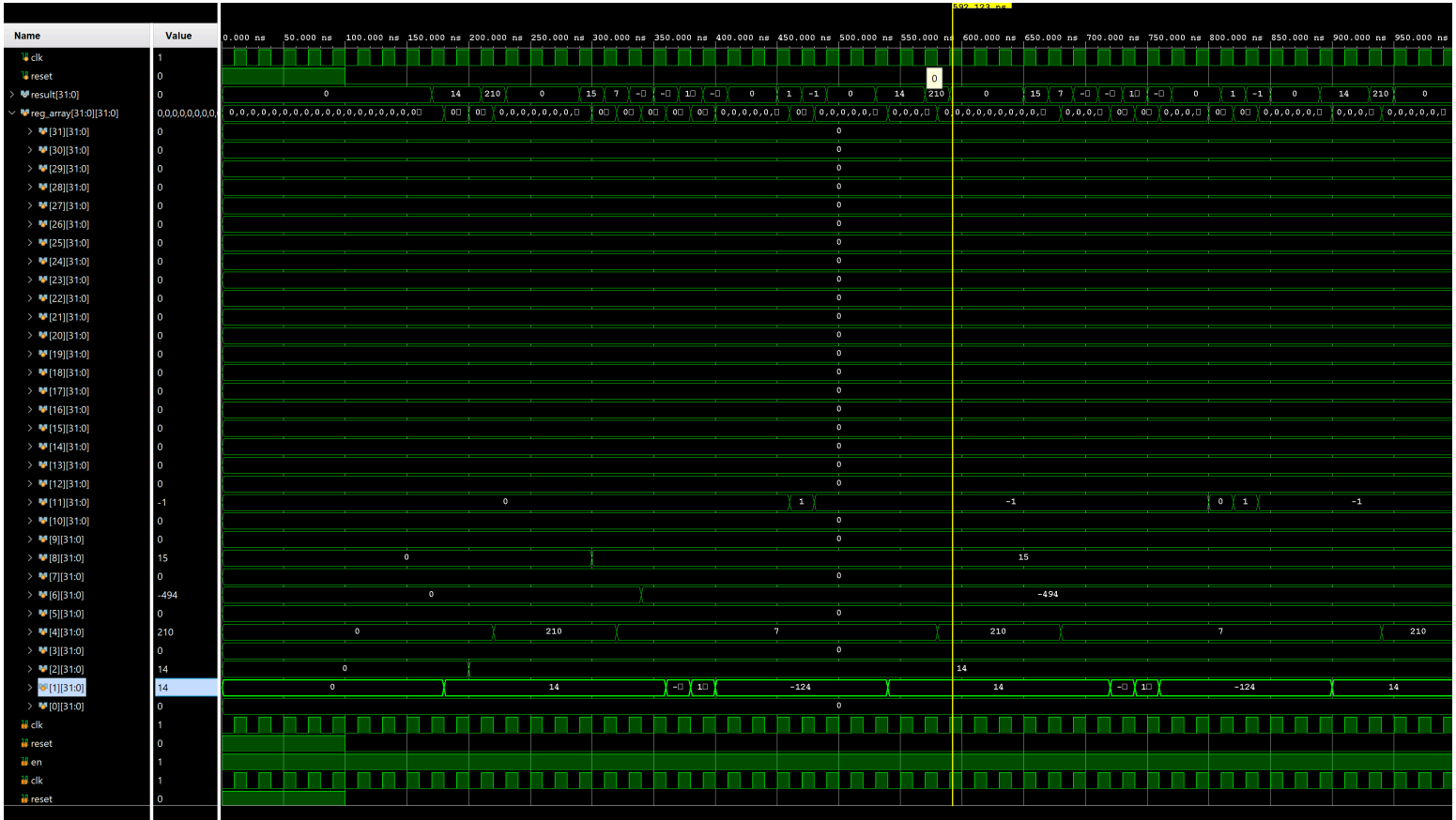
Note: Please set tb_mips_32.v as the top module when testing. Vivado sometimes incorrectly selects IF_pipe_stage.v as the default top module.

The following program was loaded from instruction memory in order to demonstrate the functionality of every instruction (with a branch taken). The program is similar to the program in the Lab 2 report, without a data dependency on the branch instruction. For a detailed explanation of each instruction please see my Lab 2 report.

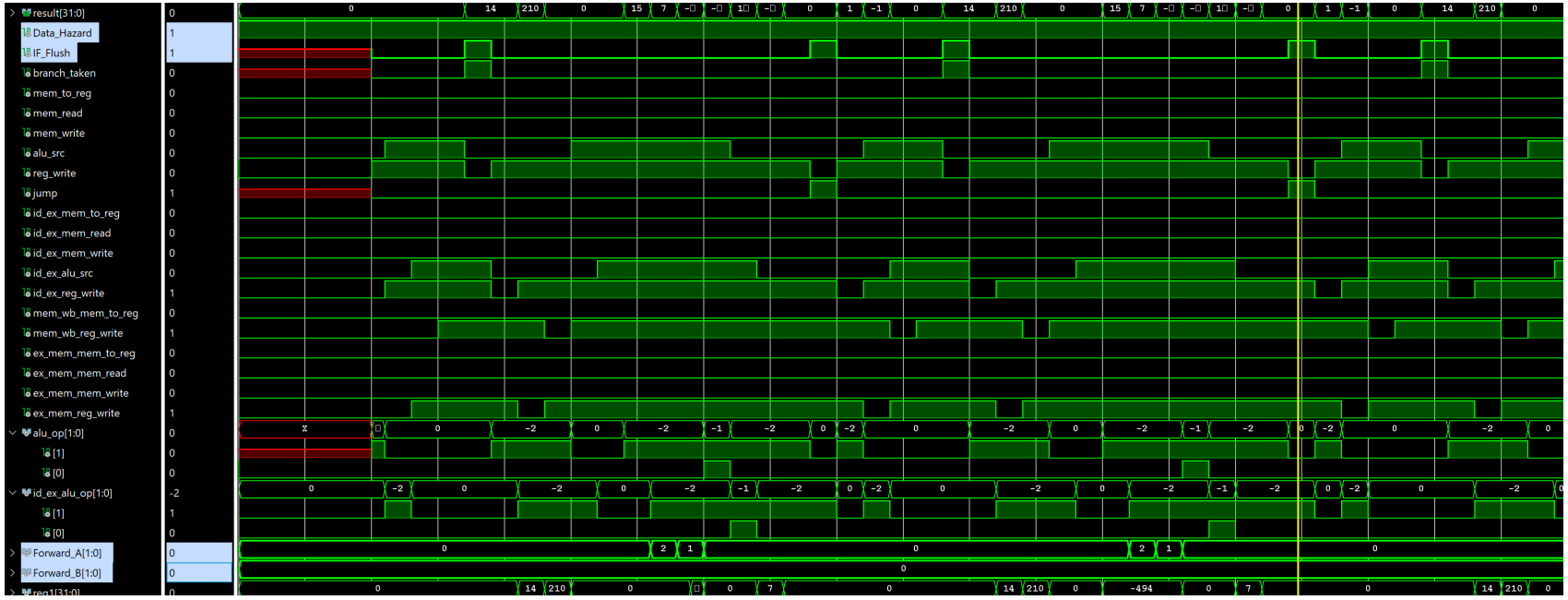
The MIPS code for this program and its corresponding waveform is shown below:

```
addi r1, r0, #14 // r1 = 14
addi r2, r0, #14 // r2 = 14
```

```
addi r4, r0, #210 // r4 = 210
beq r3, r3, #14 // taken, r3 = r3
// instructions in between branch instruction and new program counter omitted
mult r12, r1, r9 // r12 = 0
div r8, r4, r1 // r8 = 15
addi r4, r0, #7 // r4 = 7
addi r6,r0,#-494 // r6 = -494
sll r1, r6, 2 // r1 = -1976
srl r1, r6, 2 // r1 = 1073741700
sra r1, r6, 2 // r1 = -124
andi r3, r0, #78 // r3 = 0
xor r11, r9, r7 // r11 = 0
slt r11, r4, r2 // r11 = 1
nor r11, r9, r7 // r11 = -1
j #0
```

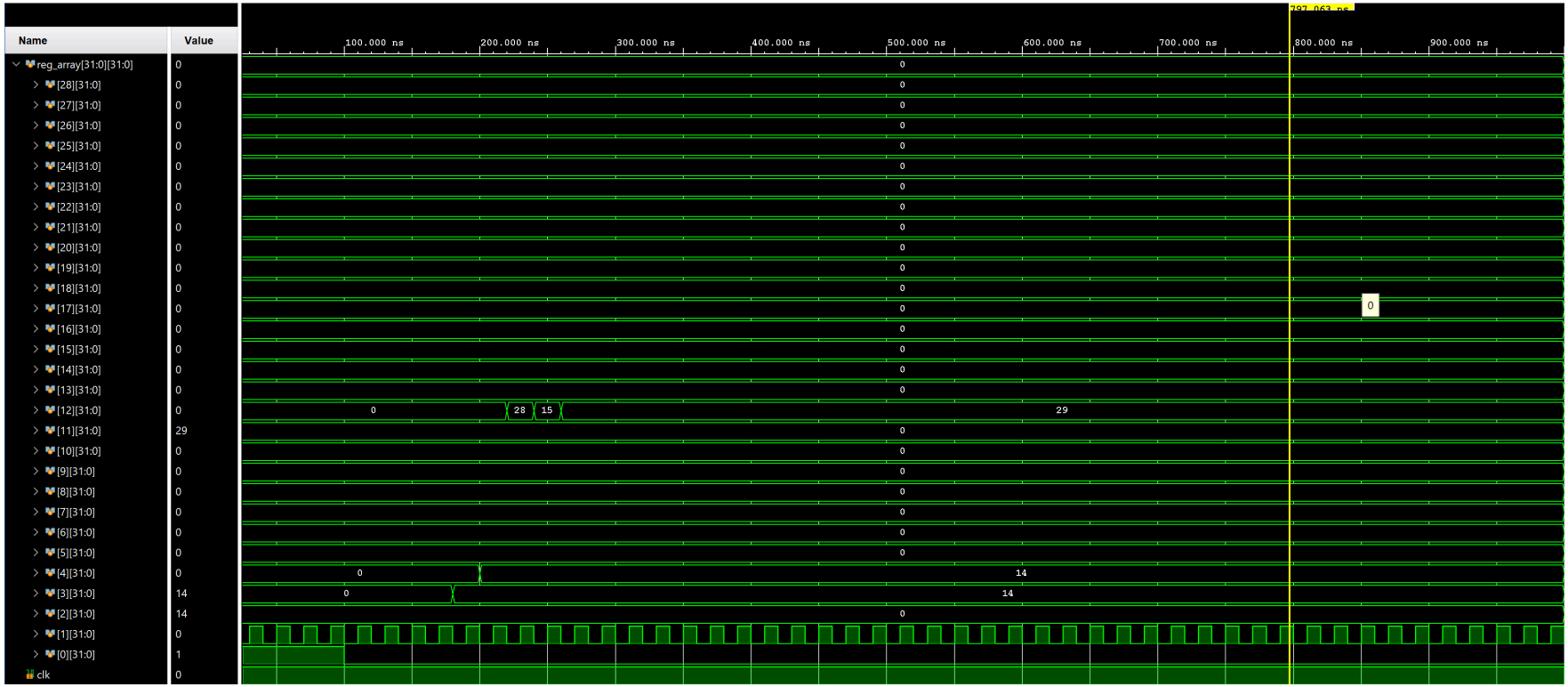


The below waveform shows the correct functionality of the Data_Hazard, IF_Flush, Forward_A, and Forward_B signals:

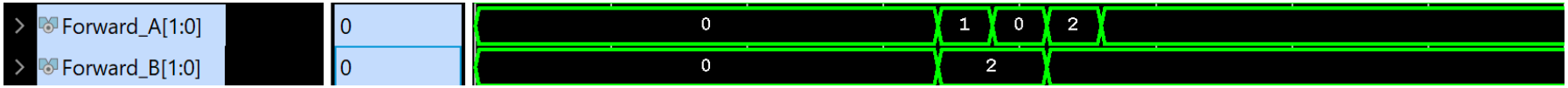


The below program shows MIPS code for a simple program that has multiple data dependencies (on the value of r10) in sequence to demonstrate the functionality of forwarding. The register array waveform is also shown below:

```
addi r1, r0, #e // r1 = 14
addi r2, r0, #e // r2 = 14
add r10, r1, r2 // r10 = 28
```




```
addi r10, r0, #f // r10 = 15
add r10, r10, r1 // r10 = 29
```



The below waveform shows the correct forwarding sequence (all else zero):

In addition to my own test programs, the provided test bench was used to test the functionality of the pipelined processor. As shown below, all outputs yield success:

Tcl Console

```
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If y
#   }
# }
# run 1000ns
xsim: Time (s): cpu = 00:00:04 ; elapsed = 00:00:05 . Memory (MB): peak = 2851.320 ; gain = 0.000
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_mips_32_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:10 . Memory (MB): peak = 2851.320 ; gain = 0.000
run all
NO DEPENDENCY ANDI  success!

NO DEPENDENCY NOR   success!

NO DEPENDENCY SLT   success!

NO DEPENDENCY SLL   success!

NO DEPENDENCY SRL   success!

NO DEPENDENCY SRA   success!

NO DEPENDENCY XOR   success!

NO DEPENDENCY MULT  success!

NO DEPENDENCY DIV   success!

ANDI No Forwarding       success!

Forward EX/MEM to EX B   success!

Forward MEM/WB to EX A  success!

SLL  No Forwarding      success!

Forward EX/MEM to EX A  success!

Forward MEM/WB to EX A  success!

XOR  No Forwarding      success!

MULT No Forwarding      success!

Forward MEM/WB to EX B  success!

DATA HAZARD RS DEPENDENCY    success!

DATA HAZARD RT DEPENDENCY   success!

CONTROL HAZARD BRANCH   success!

CONTROL HAZARD JUMP success!
```